ISSN 1330-3651 (Print), ISSN 1848-6339 (Online)
DOI: 10.17559/TV-20170324111348

A HYBRID BACKTRACKING ALGORITHM FOR AUTOMATIC TEST DATA GENERATION

Ying Xing, Yunzhan Gong, Xiaoguang Zhou, Ludi Wang, Mengke Yang, Chi Zhang, Yukun Dong

Original scientific paper
As a fundamental issue in software testing, automatic test data generation is of crucial importance, which is essentially a constraint satisfaction problem and solved by search algorithms. In our previous research, branch and bound was proposed as our constraint solver and the look-ahead methods were elaborated. Based on interval arithmetic and symbolic execution, this paper focuses on the look-back or backtracking method, which is the hybridization of forward checking and conflict-directed backjumping, with the aim of improving the efficiency of backtracking in the search procedure. The closures of variables are used to facilitate the localization of the conflicts which cause dead ends. Empirical experiments prove the effectiveness of the proposed hybrid backtracking method and its applicability in engineering.

Keywords: *automatic test data generation; backtracking; closure; conflict-directed backjumping; forward checking*

Hibridni unatražni algoritam za automatsko generiranje podataka za ispitivanje

Izvorni znanstveni članak
Kao osnovno pitanje u ispitivanju softvera, automatsko generiranje podataka za ispitivanje je od najveće važnosti, što je u biti problem zadovoljavanja ograničenja, a rješava se algoritmima pretraživanja. U našem prethodnom istraživanju za rješenje ograničenja predložena je metoda grananja i ograničavanja, a elaborirane su unaprijedne metode. Zasnovan na intervalnom aritmetičkom i simboličkom izvršenju, ovaj je rad usredotočen na unatražnu metodu, a to je povezivanje unaprijednog provjeravanja i unatražnog ispitivanja usmjerenog konflikta, u cilju poboljšanja učinkovitosti unatražnog praćenja u postupku traženja. Zatvaranja varijabli se koriste kako bi se olakšalo lociranje nesuglasica koje dovode do neriješenih rezultata (dead ends). Empirijski eksperimenti dokazuju učinkovitost predložene hibridne unatražne metode i njenu primjenljivost u inženjerstvu.

Ključne riječi: *automatsko generiranje podataka za ispitivanje; unaprijedno provjeravanje; unatražno praćenje; unatražno skakanje na osnovu konflikta; zatvaranje*

1 Introduction

As a key stage to ensure software reliability, software testing plays an indispensable part in software development. Automating software testing is a hot research topic and is of practical value to industry [1]. And being a fundamental issue in software testing, path-wise test data generation has always been a hotspot because a large number of problems in software testing can be converted into it in one way or another.

Theoretically, automatic path-wise test data generation is a constraint satisfaction problem (CSP) [2], where the path is made up of a node sequence in a control flow graph (CFG) [3]. To be exact, $X=\{x_1, x_2,\ldots, x_n\}$ is a set of variables; $R=\{r_1, r_2,\ldots, r_k\}$ is a finite set of constraints or relations between the variables along the path; $D=\{D_1, D_2,\ldots, D_n\}$ is a set of domains, and $D_i \in D$ (i=1, 2,…, n) is a finite set of possible values for x_i. For the path concerned (denoted as p), D is defined on the basis of the acceptable ranges of the variables. One solution to the problem is a set of values to instantiate each variable within its domain, denoted as $\{<x_1, V_1>,<x_2, V_2>,\ldots,<x_i, V_i>,<x_{i+1}, V_{i+1}>,\ldots,<x_n, V_n>\}(V_i \in D_i)$ to make p feasible, which means that each constraint in R should be satisfied [4].

To solve the CSP and further obtain the test data, it is required to abstract, propagate, and solve these constraints. Many researchers have made great endeavors in this field. Korel proposed an automatic test data generation method and the definition of branch function as well [5], which have inspired many researchers. Demillo and Offutt [6] put forward a technique adopting bisection and algebraic constraints to solve out the test data designed to find particular types of faults, but there was not enough heuristic information to guide the search. ADTEST [7] proposed by Gallagher et al. only considered one input variable or one predicate, and kept the solving process iterated, which was inefficient and inapplicable to real-world programs. Gupta et al. [8] found out a dynamic way of generating test data for a particular path. Robschink [9] statically converted a program into a form called Static Single Assignment (SSA), and usually resulted in systems with huge number of constraints which included irrelevant variables on some occasions. BINTEST [10] developed by Beydeda et al. guided the search procedure with bisection, which could eliminate the domains of variables that possibly included some solutions. Euclide proposed by Arnaud Gotlieb [11] was a testing tool based on constraint solving, and was employed to verify safety-critical C programs, combining symbolic and numerical analyses, constraint propagation, integer linear relaxation and search-based test data generation. Pachauri et al. [12] introduced three methods to order branches for selecting targets for coverage testing, and conducted experiments to evaluate branch ordering through memory and elitism to increase the performance of test data generation.

For the purpose of constructing an efficient constraint solving engine for automatic test data generation, we proposed best-first-search branch and bound (BFS-BB) [13] in our previous work, which is a heuristic method adopting a classical artificial intelligence algorithm, namely, branch and bound [14]. As the enhancement of BFS-BB, we optimize the part of backtracking in this paper by combining forward checking and conflict-directed backjumping, so as to increase the search efficiency. The closures of variables are used to facilitate the backtracking process.

Using BFS-BB that adopted forward checking (introduced in Section 2) as the backtracking method, we tested some real-world benchmarks in an engineering project *aa200c* available at http://www.moshier.net/, and it was found that the paths with backtracking accounted for 32 % of all the paths, whereas the searching time consumed on them accounted for 81 % of the total time consumption. The efficiency of backtracking is a key problem that influences the performance of the test data generation method. Therefore, we need to design a more efficient backtracking method for BFS-BB, and we also attempt to make evaluations on the influence of our method on the efficiency of backtracking empirically.

The remainder of this paper is organized as follows. Section 2 introduces the relevant concepts of backtracking. In Section 3, we elaborate the hybrid backtracking algorithm and its backtracking strategies. Section 4 conducts experiments to analyze and evaluate the proposed backtracking strategies. Conclusion of this paper and directions for future research are presented in Section 5.

2 Backtracking search

The methods used to solve CSPs are usually based on backtracking search [15]. In summary, a backtracking algorithm finds the solution by extending partial problem solutions. There is a current partial solution which the backtracking algorithm attempts to extend at every search stage and finally becomes the full solution. In the search procedure, variables are divided into three sets: past variables (*PV*, already instantiated), current variable (being instantiated currently), and future variables (*FV*, not yet instantiated). An inconsistency is a consistent partial solution composed of the instantiations of past variables. The partial solution cannot be added any more variables and cannot be part of the full solution. A dead end is encountered after all the values in the domain of the current variable have been tried out. Under this situation, some instantiated variables will become uninstantiated, which means they are eliminated from the current partial solution.

The techniques used to improve a search algorithm are classified as look-ahead and look-back methods. Look-ahead methods take effect every time the search is ready to extend the current partial solution, and look-back methods take effect whenever a dead end occurs and the search is ready for the backtracking step. The look-ahead methods in BFS-BB were elaborated in our previous work [13]. In this paper, the look-back methods are our focus, which include the functions that determine which variable to backtrack to by analyzing the reason for the dead end and decide what new constraints to record so that the same conflict will not appear again in the search [16].

Chronological backtracking (BT) [17] is the simplest yet the most widely used backtracking algorithm, as it chronologically backtracks to the variable last instantiated when a dead end occurs. Rather than backtracking to the last instantiated variable chronologically, backjmping (BJ) [18] goes to the deepest past variable in conflict with the current variable. In conflict-directed backjumping (CBJ) [19], each variable has its conflicting set consisting of the past variables which failed consistency checks with its current instantiation. Unlike the above backward methods, forward checking (FC) [20] conducts consistency check forward, namely, the check is between the current variable and future variables. BFS-BB adopted FC as the backtracking strategy. The hybrid algorithm forward checking and conflict-directed backjumping (FC-CBJ) draws on the advantages of both FC and CBJ, and was proved to be the champion among common backtracking algorithms [19].

3 The hybrid backtracking strategy

As mentioned in Section 2, FC-CBJ performs more efficiently than other backtracking algorithms, and we adopt FC-CBJ as our backtracking algorithm in this paper. Interval arithmetic [21] and symbolic execution [3, 22] are adopted to facilitate the process of FC-CBJ. Besides, the closures of variables are proposed to localize the conflict more precisely. We elaborate FC and CBJ in Sections 3.1 and 3.2, respectively. The hybrid searching algorithm is proposed in Section 3.3.

3.1 Forward checking

Forward checking is accomplished by interval arithmetic. To better explain the procedure, we define branch condition as below.

Definition 1. Let B be the set of $\{true, false\}$ and D^a be the set of the domains of all the variables before the ath branch, if there are k branches along the path, the **branch condition** $Br(n_{qa}, n_{qa+1})$: $D^a \rightarrow B$ ($a \in [1, k]$) where n_{qa} is a branching node is computed by Eq. (1).

$$Br(n_{qa}, n_{qa+1}) = \begin{cases} true, D^a \cap \widetilde{D}^a \neq \varnothing \\ false, D^a \cap \widetilde{D}^a = \varnothing \end{cases} \tag{1}$$

In Eq. (1), D^a meets the previous a-1 branch conditions and will be input to compute the ath branch condition. $\widetilde{D}^a$ which is a set of volatile domains is the result of computing $Br(n_{qa}, n_{qa+1})$ with D^a, and meets the ath branch condition. $D^a \cap \widetilde{D}^a \neq \varnothing$ means that $D^a \cap \widetilde{D}^a$ satisfies all the previous a-1 branch conditions and the ath branch condition simultaneously, so that interval arithmetic can proceed to compute the remaining branch conditions, whereas $D^a \cap \widetilde{D}^a = \varnothing$ means that the domain of at least one variable (which may be a future variable) is annihilated and R is not met. For the k branches along the path, all the k branch conditions should be *true* to make R met. This process is demonstrated by Fig. 1.

Accurately, the input of interval arithmetic is the set of the domains of all the variables represented as D^1, and the branch condition corresponding to the branch (n_{q1}, n_{q1+1}) where n_{q1} is the first branching node evaluated. Typically, the branch condition $Br(n_{q1}, n_{q1+1})$ cannot be satisfied by all the values in D^1, but some values in a subset $D^2 \subseteq D^1$ ensuring the traversal of the branch (n_{q1}, n_{q1+1}), i.e., $D^1 \xrightarrow{Br(n_{q1}, n_{q1+1})} D^2$. Next the branch condition $Br(n_{q2}, n_{q2+1})$ is evaluated by the set of the domains of all the variables (D^2). Again, generally $Br(n_{q2}, n_{q2+1})$ is only satisfied by a subset $D^3 \subseteq D^2$. This process carries on along p until all the branch conditions are satisfied and D^{k+1} is

returned as the set of the domains of all the variables. The process is the propagation of the branch conditions along p in the form of $D^1 \xrightarrow{Br(n_{q1},n_{q1+1})} D^2 \xrightarrow{Br(n_{q2},n_{q2+1})} D^3 \ldots D^k \xrightarrow{Br(n_{qk},n_{qk+1})} D^{k+1}$, where $D^1 \supseteq D^2 \supseteq D^3 \ldots \supseteq D^k \supseteq D^{k+1}$, shown as Fig. 1(a). But if in this procedure $Br(n_{qh}, n_{qh+1})$ ($1 \le h \le k$) is *false*, indicating the detection of a conflict, then interval arithmetic is terminated, shown as Fig. 1(b). In our method, the process of calculating each branch condition is considered to be a constraint check, which is a coarse-grained checking manner compared with some arc consistency checking algorithms, for example, AC-3[23]. It can be found that in the procedure of interval arithmetic, it is always the set of the domains of all the variables that is involved in the computation, which ensures possible domain reduction or annihilation of all the variables including those in FV.

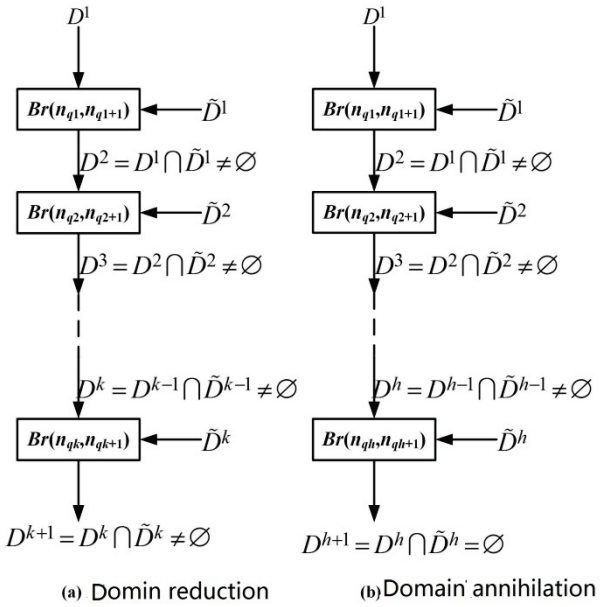

Figure 1 The conflict detecting process of interval arithmetic

For convenience, we suppose that the instantiation of variables is in the predefined order: $x_1, x_2, \ldots, x_n$. But in the actual implementation, dynamic ordering is adopted, in which the state of the search determines the variable to instantiate next. Assuming that the current variable is x_i, the input of forward checking is the set of the domains of all the variables represented as $\{[V_1, V_1], [V_2, V_2], \ldots, [V_i, V_i], D_{i+1}, \ldots, D_n\}$, which includes three parts. The domains of the past variables are all fixed values with the same top and bottom value which have been verified consistent by forward checking; the domain of the current variable is a fixed value with the same top and bottom value which is being verified by forward checking; the domains of future variables are basically a range of values which have been filtered by the past instantiations and will be filtered by the current instantiation. Since the main purpose of forward checking is to judge whether the assignment V_i for the current variable x_i will lead to an inconsistency or possible domain annihilation, we just use *forward check* (V_i) to denote it. In our previous research [24], interval arithmetic is iterated in order to increase the precision of forward checking.

3.2 Conflict-directed backtracking

A backtracking search can be regarded as traversing a search tree, and we give the definition of level.

Definition 2. The **level** of a variable marks its order of instantiation in the search process. It is exactly the number of instantiated variables. For example, level 0 marks the root of the search tree, and no variable has been instantiated; level 1 marks x_1 which is the first variable to be instantiated, and the like. The levels nearer to the root are shallower levels, and the levels farther from the root are deeper levels.

Fig. 2 clearly shows the so-called CBJ, which is taken between two variables that are not on neighboring levels. Since the backjumping is directed by conflict, the following are some definitions related to conflict.

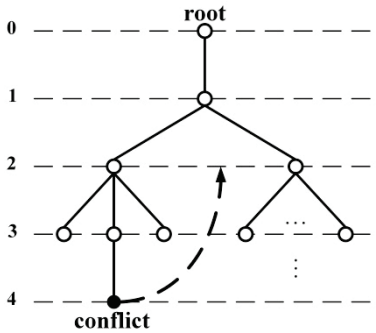

Figure 2 An example of CBJ

Definition 3. If two variables x_i and x_j have relation $r_h \in R$ ($h \in [1, k]$), then x_i and x_j are **directly related variables** to each other. The set of the directly related variables of x_i is denoted as $S_{rel}(x_i)$. In terms of test data generation, the variables in the same predicate are regarded as directly related variables to each other.

Definition 4. The set of the variables that can be derived from any relation to x_i both directly and indirectly forms the **closure** of x_i, denoted as $C(x_i)$, which is calculated iteratively by Eq. (2).

$$C(x_i) = S_{rel}(x_i) \cup \bigcup_{x_j \in S_{rel}(x_i)} S_{rel}(x_j) \tag{2}$$

The closure of a variable is a data structure storing the variable and the variables that have both direct and indirect relations with it: a mapping associating a variable with the variables bounded by all the relations along the path. All the variables mentioned in this paper are symbolic variables, and a simple example as shown in Fig. 3 explains closure.

```
void test(int x1, int x2, int x3, int x4,
   int x5, int x6, int x7, int x8)
{x1=x2+1;
 x3=2*x5;
 if(x1>x3)
   if(x3+x4==100)
     if(x5-x7<30)
       if(x6+x8<100)
         printf( "example" );
}
```

Figure 3 The program test

If we attempt to generate test data for the path passing all the *if* statements, then we have Tab. 1 about the symbolic variable and closure of each variable.

Table 1 The symbolic variable and closure corresponding to each variable in program *test*

Variable	Symbolic variable	Closure
$x1$	$x2+1$	Ø
$x2$	$x2$	{ $x2$, $x4$, $x5$, $x7$}
$x3$	$2*x5$	Ø
$x4$	$x4$	{$x2$, $x4$, $x5$, $x7$}
$x5$	$x5$	{$x2$, $x4$, $x5$, $x7$}
$x6$	$x6$	{$x6$, $x8$}
$x7$	$x7$	{$x2$, $x4$, $x5$, $x7$}
$x8$	$x8$	{$x6$, $x8$}

Based on the above analysis, it can be concluded that if any past instantiation is responsible for the current inconsistency, that instantiation must have been launched to one or more variables in the same closure as the current variable, and the following is the definition of conflicting variable.

Definition 5. If the current instantiation $<x_i, V_i>$ causes an inconsistency, and there is a set of past variables in the same closure as x_i and the corresponding domains are annihilated, then the deepest variable x_j in this set is the **conflicting variable** of the current inconsistency.

3.3 Hybrid backtracking algorithm

Since we combine FC and CBJ (adopting the closures of variables) as a hybrid backtracking algorithm, it is referred to as BFS-BB-Hybrid Backtracking (BFS-BB-HB) in this paper. Some notations in BFS-BB-HB are explained by Tab. 2.

Table 2 Some notations used in BFS-BB-HB

Variable	Meaning
x^*	The current variable
V^*	The assignment to x^*
$S_{conflict}(x^*)$	The conflicting set of x^*
$x_{level(x^*-1)}$	The variable instantiated just ahead of x^*
x_f	The conflicting variable

Algorithm. BFS-BB-HB
Input p : the path to be covered
Output *result*$\{<x_1,V_1>,<x_2,V_2>,\ldots,<x_n,V_n>\}$: test data making p feasible
Begin
1: $result \leftarrow null$;
2: $S_{rel}(x_i) \leftarrow \emptyset$; ($i\in[1,n]$)
3: **for** $m\rightarrow 1$: k
4: **if**($x_i\ r_m\ x_j$) ($i, j\in[1,n]$)
5: $S_{rel}(x_i) \leftarrow S_{rel}(x_i) \cup \{x_j\}$;
6: $S_{rel}(x_j) \leftarrow S_{rel}(x_j) \cup \{x_i\}$
7: $C(x_i) \leftarrow S_{rel}(x_i)$; ($i\in[1,n]$)
8: **for** $i\rightarrow 1$: n
9: **if** ($x_a\ r_h\ x_b$)($x_a \notin C(x_i) \wedge x_b\in C(x_i)$, $h\in[1,k]$, $r_h\in R$)
10: $C(x_i) \leftarrow C(x_i) \cup \bigcup_{x_p\in S_{rel}(x_a)} S_{rel}(x_p)$;
11: **while**($FV\neq \emptyset$)
12: $x^*\leftarrow$sort (FV);
13: $V^*\leftarrow$select (D^*);
14: $S_{conflict}(x^*)\leftarrow\emptyset$;
15: forward check (V^*);
16: **while** (forward checking fails)
17: **foreach** $x_j\in C(x^*)$
18: **if** ($D_j = \emptyset \wedge x_j\in PV$)
19: $S_{conflict}(x^*)\leftarrow S_{conflict}(x^*)\cup \{x_j\}$;
20: **if**($S_{conflict}(x^*)\neq \emptyset$)
21: $x_f\leftarrow$quicksort ($S_{conflict}(x^*)$);
22: **else** $x_f\leftarrow x_{level(x^*-1)}$;
23: $PV\leftarrow PV-\{x_f\}$;
24: $x^* \leftarrow x_f$;
25: $V^*\leftarrow$select after conflict (D^*);
26: forward check (V^*);
27: $result \leftarrow result+\{<x^*, V^*>\}$;
28: $FV\leftarrow FV-\{x^*\}$;
29: $PV\leftarrow PV+\{x^*\}$;
30: **return** *result*;
End

The path concerned is the input of BFS-BB-HB, including the constraints to be met (R), the set of input variables (X), and the domains corresponding to the variables (D). The test data (*result*) is null. First, the set of related variables and closure are calculated for each variable as shown by lines 2-10. When FV is not an empty set, which means there are still variables uninstantiated, the following steps are taken. The variables in FV are sorted to determine the current variable x^*, and a value V^* is selected from the domain of x^*(D^*), which are shown by lines 12 and 13, and have been elaborated in our previous work [25]. $S_{conflict}(x^*)$ is initialized as shown by line 14. Forward checking (line 15) is conducted to judge whether V^* for x^* will cause any inconsistency and reduce the domains of future variables. If forward checking succeeds, then $<x^*,V^*>$ is added to *result*, and x^* is removed from FV and put into PV, as shown by lines 27-29. Then the ordering of FV begins. If forward checking fails, indicating that an inconsistency is caused, then the conflicting variable should be determined. The past variables with annihilated domains are put into the set of the conflicting variables of x^*, and quicksort is used to find the deepest variable (x_f), which is the conflicting variable. In that case, CBJ will directly go to the level of x_f as shown by lines 17-21. In some cases, the conflicting set of x^* is empty, then BT is inevitable and the variable on the level just above x^* becomes the current variable (line 22). As shown by lines 23-25, the conflicting information is adopted to select a value for the current variable, which is introduced in our previous work [25]. And another forward checking begins (line 26) until FV becomes empty. Finally, *result* is returned as the test data that makes the path feasible as shown by line 30.

During BFS-BB-HB search, a variable may exist in any of the four sets, which are future, current, past, and conflicting, depending on the different search operations taken on it, as shown by Fig. 4. At the very beginning of the search, all the variables are in FV, whereas when the search ends, all the variables are in PV. The transformation of variables is also illustrated by Fig. 4.

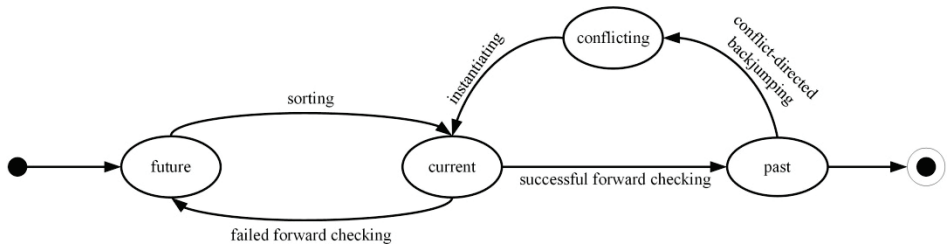

Figure 4 The transformation of the states of variables in the search process

4 Experimental analyses and empirical evaluations

To test the effectiveness of BFS-BB-HB, we carried out a large number of experiments. The paths to be covered were provided by CTS. The algorithms were implemented in Java and run on the platform of Java Runtime Environment (JRE). Since closure is an important factor in our backtracking strategy, in Section 4.1 experiments were made to evaluate the effectiveness of our method in terms of generation time and the number of backtracking for varying numbers of closures. In Section 4.2, the hybrid backtracking method FC-CBJ (adopted by BFS-BB-HB) and the chronologically backtracking method FC (adopted by BFS-BB) were compared, and the test beds were some CSP problems. BFS-BB-HB was used to test an engineering project in Section 4.3.

4.1 Testing different numbers of closures

First, we evaluated the influence of the closures of variables on the performance of test data generation. This was made by repeatedly running BFS-BB-HB on generated test programs, each of which included 10 input variables. Using statement coverage, in each test the program had 10 variables and n ($n \in [1, 10]$) *if* statements, and there was only one path to be covered, which is made up of branches totally *true*, i.e., all the expressions were in the same form as the corresponding predicates. Each expression contained all the 10 variables, which may be in different closures. We tried to make each closure contain roughly the same number of variables, which means when there were 1, 2, and 5 closures, each closure contained the same number of variables, namely, 10, 5, and 2, respectively; while when there were 3 closures, the numbers of variables in each closure were 3, 3, and 4, respectively, and those numbers were 2, 2, 3, 3 for 4 closures. Take 2 closures for an example, each *if* statement was an expression in the form of Eq. (3).

$$([a_{n1}, a_{n1}, ..., a_{n5}][x_1, x_2, ..., x_5]' \ rel\_op \ const[n][1]) \wedge ([a_{n6}, a_{n7}, ..., a_{n10}][x_6, x_7, ..., x_{10}]' \ rel\_op \ const[n][2]) \quad (3)$$

In Eq. (3), a_{n1}, a_{n2}, … ,a_{n10} (n=1,2,…,10) were numbers generated by random function, $rel\_op \in \{>,\geq,<,\leq,=,\neq\}$, and $const[n][2]$ was an array composed of constants generated by random function. The randomly generated a_{ni} (i=1, 2,…, 10), $const[n][1]$, and $const[n][2]$ were selected to make the path feasible. Thus, the linear relation between the variables in the same closure was constructed in the strongest manner. The programs were each tested 100 times, and the average time required to generate the test data and the average number of backtracking for each test were recorded. The environment that the experiments were performed in was Windows 7 with 32-bits, Pentium 4 with 3.8 GHz and 4 GB memory. The result of the comparison is shown by Fig. 5 and Fig. 6.

From Fig. 5 (a), it can be found that for a fixed number of closures, average generation time increased with the number of expressions, which was more obvious when there were 6-10 expressions in the programs under test (PUTs). The reason is that the complexity of constraints increased with the number of expressions, and the search was basically backtrack free when there were not too many constraints. Fig. 5 (b) shows that for the same number of expressions, generation time decreased with the number of closures, because more closures meant less variables involved in each constraint, reducing the complexity of the search. This was also true of the relationship between average number of backtracking and the number of closures, which is shown by Fig. 6.

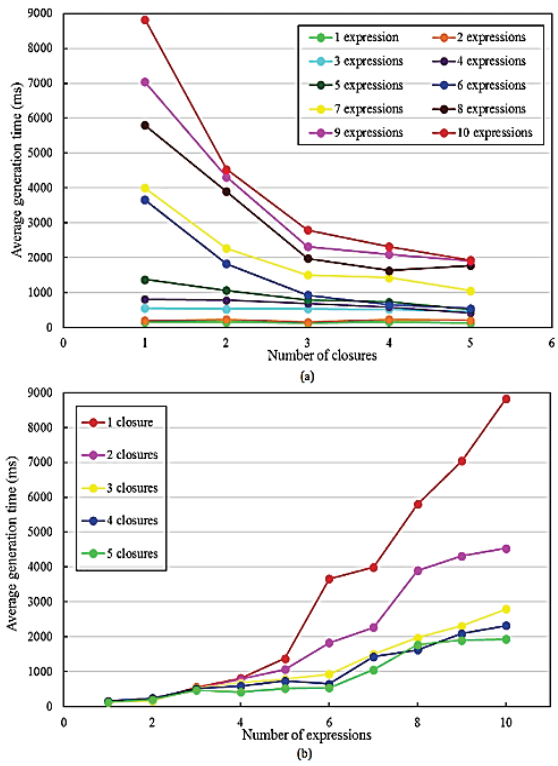

Figure 5 The relationship between average generation time and the number of closures

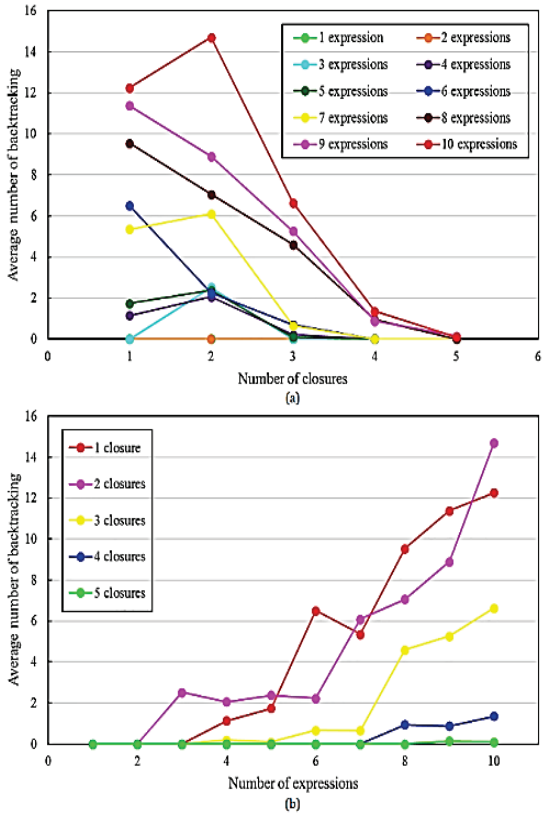

Figure 6 The relationship between average number of backtracking and the number of closures

4.2 Comparison between BFS-BB-HB and BFS-BB

We also carried out comparison experiments between the method proposed in this paper (BFS-BB-HB) and the method proposed in [13] (BFS-BB). The environment that the experiments were performed was Windows 7 with 32-bits, Pentium 4 with 3.00 GHz and 4 GB memory. Some CSP problems from http://www.csplib.org/Problems/ were used as the test beds. A step backward in the search tree shows that the backtracking operation is taken once.

The first selected CSP problem was the n-queens problem. For each test bed (n=4,5,…,9), the experiments were conducted 100 times, and average number of backtracking and average time consumption were adopted for comparison. The testing result is shown in Tab. 3, which was in accordance with the distribution of the solutions to the n-queens problem when n varies from 4 to 9 as shown by Tab. 4, where we can find that the solutions to 6-queen are less than those to 5-queen. And since 1-queen is self-evident, and there are no solutions for 2-queen and 3-queen, our experiments started with 4-queen. For all the tests, BFS-BB-HB performed better than BFS-BB as shown in bold, and our hybrid backtracking strategy functioned well for n-queens problem.

Table 3 The comparison result of testing the n-queens problem using BFS-BB-HB and BFS-BB

N	Average number of backtracking		Average time consumption (ms)	
4	**0,67**	0,79	**132,15**	143,48
5	**0**	0	**142,30**	185,84
6	**2,84**	4,72	**1233,77**	2699,03
7	**0,56**	0,57	**942,74**	1012,63
8	**2,86**	4,13	**3417,05**	4604,20
9	**4,27**	4,38	**5636,65**	5743,14

Table 4 The number of solutions for n-queens problem when n varies from 1 to 9

N	1	2	3	4	5	6	7	8	9
Number of solutions	1	0	0	2	10	4	40	92	352

The following are two other selected CSP problems, which are magic squares and sequences (n=4) and magic hexagon. The experiments were carried out 100 times for each test bed, with average number of backtracking, average number of constraint checks and average time consumption recorded for comparison. Tab. 5 presents the experimental results, showing that BFS-BB-HB outperformed BFS-BB for all the cases. For the number of backtracking, BFS-BB-HB took 12 % and 42 % of BFS-BB respectively, as shown in bold in column 4. For the number of constraint checks, BFS-BB-HB accounted for 24 % and 48 % of BFS-BB respectively, as shown in bold in column 7. For time consumption, BFS-BB-HB occupied 21 % and 47 % of BFS-BB respectively, as shown in bold in column 10.

In short, BFS-BB-HB performed well for the selected CSP problems, especially in the improvement of backtracking efficiency.

Table 5 The result of testing two CSP problems

Problem	Average number of backtracking			Average number of constraint checks			Average time consumption (ms)		
	BFS-BB-HB	BFS-BB	Ratio	BFS-BB-HB	BFS-BB	Ratio	BFS-BB-HB	BFS-BB	Ratio
Magic squares and sequences (n=4)	5,36	43,98	**12 %**	10 305,42	42 526,50	**24 %**	14 999,66	71 495,28	**21 %**
Magic hexagon	0,27	0,65	**42 %**	3573,94	7379,56	**48 %**	4659,56	9994,08	**47 %**

4.3 Testing a project in engineering

In this part, we made experiments using BFS-BB-HB to test engineering projects. The experiments adopted statement coverage, and were performed in the environment of Windows 7 with 32-bits, Intel Pentium (R) G640 with 2.80 GHz and 2 GB memory. The PUTs were from *aa200c* mentioned in Section 1, and the selected ones contained diversified data structures and types. The result is shown in Tab. 6. Constraint check was elaborated in Section 3, and it was used to evaluate forward checking of BFS-BB-HB. Two conclusions can be drawn from Tab. 6. First, BFS-BB-HB performed better when there were not pointers in the PUTs. We should exert more efforts to strengthen the capability of handling pointers. Second, since the number of variables in expressions will influence the search efficiency [24], the same number of constraint checks may lead to different time consumption. Generally speaking, BFS-BB-HB performed within acceptable time, but it still needs optimization.

Table 6 The result of testing aa200c by BFS-BB-HB

File	Function	Types of variables	Paths from CTS	Average number of constraint checks	Average time consumption (ms)
angles	angles	double[], double[], double[]	path1	10	313,74
deflec	relativity	double[], double[], double[]	path1	4	34,30
			path2	11	151,54
diurab	diurab	double, double *, double *	path1	16	207,98
			path2	16	188,56
diurpx	diurpx	double, double *, double *, double	path1	5	353,46
			path2	5	17,40
fk4fk5	fk4fk5	double[], double[], struct star *	path1	30	1034,88
			path2	15	828,90
			path3	30	1146,44
			path4	15	1018,78
			path5	15	935,74
lightt	lightt	struct orbit *, double[], double[]	path1	3	68,48
			path2	3	87,92
			path3	8	616,5
			path4	6	215,04
			path5	4	101,04
			path6	6	224,14
lonlat	lonlat	double[], double, double[], int	path1	18	366,48
			path2	18	250,18
			path3	20	349,72
pctomot	main	int, char **	path1	71,40	1403,18
			path2	15	68,12

5 Conclusion and future work

The need for automating the testing procedure has become increasingly urgent in recent years. And as a fundamental issue in software testing, path-wise test data generation is of particular significance. We put forward an intelligent constraint solver and elaborated the look-ahead methods in our previous research. In this paper, we studied the look-back methods in detail, which function when a dead end occurs. The proposed backtracking method hybridized forward checking and conflict-directed backjumping, and the closures of variables were introduced in the process of detecting conflict. The testing of some CSP problems and some engineering benchmarks proved the effectiveness of the hybrid backtracking method and its applicability in engineering.

Our future research will continue to increase the backtracking efficiency as well as enhance the effectiveness of the test data generation method. Parallel computing among different closures will be touched upon in our future research.

Acknowledgements

This work was supported by the National Grand Fundamental Research 863 Program of China (No. 2012AA011201), the Major Program of the National Natural Science Foundation of China (No. 91318301), the National Key Research and Development Program of China (No. 2016YFC0803206), the Fundamental Research Funds for the Central Universities, the National Nonprofit Industry Research (No. 201313009-08), and the Foundation for Outstanding Young Scientist in Shandong Province (No. BS2015DX017).

6 References

[1] Elsayed, E. A. Overview of reliability testing. // IEEE Transactions on Reliability. 61, 2(2012), pp. 282-291. DOI: 10.1109/TR.2012.2194190

[2] Kasprzak, W.; Szynkiewicz, W.; Zlatanov, D.; Zielinska, T. A hierarchical CSP search for path planning of cooperating self-reconfigurable mobile fixtures. // Engineering Applications of Artificial Intelligence. 34, (2014), pp. 85-98. DOI: 10.1016/j.engappai.2014.05.013

[3] Yang, G. W.; Person, S.; Rungta, N.; Khurshid, S. Directed incremental symbolic execution. // ACM Transactions on Software Engineering and Methodology. 24, 3 (2014). DOI: 10.1145/2629536

[4] Cooper, M. C.; Duchein, A.; Mouelhi, A. E.; Escamocher, G. Broken triangles: From value merging to a tractable class of generality constraint satisfaction problems. // Artificial Intelligence. 234, (2016), pp. 196-218. DOI: 10.1016/j.artint.2016.02.001

[5] Korel, B. Automated Software Test Data Generation. // IEEE Transactions on Software Engineering. 16, 8(1990), pp. 870-879. DOI: 10.1109/32.57624

[6] DeMillo, R. A.; Offutt, A. J. Constraint-based automatic test data generation. // IEEE Transactions on Software Engineering. 17, 9(1991), pp. 900-910. DOI: 10.1109/32.92910

[7] Gallagher, M. J.; Narasimhan, V. L. ADTEST: A test data generation suite for Ada software systems. // IEEE Transactions on Software Engineering. 23, 8(1997), pp. 473-484. DOI: 10.1109/32.624304

[8] Gupta, N.; Mathur, A. P.; Soffia, M. L. UNA based iterative test data generation and its evaluation. // Proceedings of the 14th IEEE Int. Conference on Automated Software Engineering. / Cocoa Beach, 1999, pp. 224-232. DOI: 10.1109/ase.1999.802270

[9] Robschink, T.; Snelting, G. Efficient path conditions in dependence graphs. // Proceedings of the 24th Int. Conference on Software Engineering. / Orlando, 2002, pp. 478-488. DOI: 10.1145/581396.581398

[10] Beyleda, S.; Gruhn, V. BINTEST-Binary Search-based Test Case Generation. // Proceedings of the 27th Annual Int. Conference on Computer Software and Applications. / Dallas, 2003, pp. 28-33. DOI: 10.1109/cmpsac.2003.1245318
[11] Gotlieb, A. Euclide: A Constraint-Based Testing Framework for Critical C Programs. // Proceedings of the 9th. Int. Conference on Software Testing Verification and Validation. / Denver, 2009, pp. 151-160. DOI: 10.1109/icst.2009.10
[12] Pachauri, A.; Srivastava, G. Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism. // Journal of Systems and Software. 86, 5(2013), pp. 1191-1208. DOI: 10.1016/j.jss.2012.11.045
[13] Xing, Y.; Gong Y. Z.; Wang, Y. W.; Zhang X. Z. Path-wise Test Data Generation Based on Heuristic Look-ahead Methods. // Mathematical Problems in Engineering. 2014. URL:https://www.hindawi.com/journals/mpe/2014/642630/ (05.14.2014). DOI: 10.1155/2014/642630
[14] Vilà, M.; Pereira, J. A branch-and-bound algorithm for assembly line worker assignment and balancing problems. // Computers and Operations Research. 44, (2014), pp. 105-114. DOI: 10.1016/j.cor.2013.10.016
[15] Chang, L.; Qin, F.; Li, A. A novel backtracking scheme for attitude determination-based initial alignment. // IEEE Transactions on Automation Science and Engineering. 12, 1(2015), pp. 384-390. DOI: 10.1109/TASE.2014.2346581
[16] Dechter, R.; Meiri, I.; Pearl, J. Temporal constraint networks. // Artificial Intelligence. 49, 1(1991), pp. 61-95. DOI: 10.1016/0004-3702(91)90006-6
[17] Bitner, J. R.; Reingold, E. M. Backtrack programming techniques. // Communications of the ACM. 18, 11(1975), pp. 651-656. DOI: 10.1145/361219.361224
[18] Collavizza, H.; Vinh, N. L.; Ponsini, O.; Rueher, M.; Rollet, A. Constraint-based BMC: a backjumping strategy. // International Journal on Software Tools for Technology Transfer. 16, 1(2014), pp. 103-121. DOI: 10.1007/s10009-012-0258-6
[19] Prosser, P. Hybrid algorithms for the constraint satisfaction problem. // Computational. Intelligence. 9, 3(1993), pp. 268-299. DOI: 10.1111/j.1467-8640.1993.tb00310.x
[20] Haralick, R. M.; Elliott, G. L. Increasing tree search efficiency for constraint satisfaction problems. // Artificial Intelligence. 14, 3(1980), pp. 263-313. DOI: 10.1016/0004-3702(80)90051-X
[21] Pereira, D. R.; Papa, J. P. A New Approach to Contextual Learning using Interval Arithmetic and its Applications for Land-use Classification. // Pattern Recognition Letters. 13, 7(2016), pp. 1-7. DOI: 10.1016/j.patrec.2016.03.020
[22] Delahaye, M.; Botella, B.; Gotlieb, A. Infeasible path generalization in dynamic symbolic execution. // Information and Software Technology. 58, (2015), pp. 403-418. DOI: 10.1016/j.infsof.2014.07.012
[23] Mackworth A. K. 1977. Consistency in networks of relations. // Artificial Intelligence. 8, 1(1977), pp. 99-118. DOI: 10.1016/0004-3702(77)90007-8
[24] Xing, Y.; Gong, Y. Z.; Wang, Y. W.; Zhang, X.Z. The application of iterative interval arithmetic in path-wise test data generation. // Engineering Applications of Artificial Intelligence. 45, (2015), pp. 441-451. DOI: 10.1016/j.engappai.2015.07.021
[25] Xing, Y.; Gong, Y. Z.; Wang, Y. W.; Zhang, X. Z. A hybrid intelligent search algorithm for automatic test data generation. // Mathematical Problems in Engineering. 2015. URL:https://www.hindawi.com/journals/mpe/2015/617685/ (06.28.2015). DOI: 10.1155/2015/617685

Authors' addresses

Ying Xing, lecturer
Automation school,
Beijing University of Posts and Telecommunications
No. 10, Xitucheng Road, Haidian District, Beijing, China
E-mail: lovelyjamie@yeah.net

Yunzhan Gong, professor
State Key laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications
No. 10, Xitucheng Road, Haidian District, Beijing, China
E-mail: gongyz@bupt.edu.cn

Xiaoguang Zhou, professor
Automation school,
Beijing University of Posts and Telecommunications
No. 10, Xitucheng Road, Haidian District, Beijing, China
E-mail: zxg@bupt.edu.cn

Ludi Wang
Automation school,
Beijing University of Posts and Telecommunications
No. 10, Xitucheng Road, Haidian District, Beijing, China
E-mail: 15117928901@163.com

Mengke Yang
Automation school,
Beijing University of Posts and Telecommunications
No. 10, Xitucheng Road, Haidian District, Beijing, China
E-mail: 15201646681@139.com

Chi Zhang
Automation school,
Beijing University of Posts and Telecommunications
No. 10, Xitucheng Road, Haidian District, Beijing, China
E-mail: zcsnail@gmail.com

Yukun Dong, lecturer
College of Computer & Communication Engineering,
China University of Petroleum
No. 66, Changjiang West Road, Huangdao District,
Qingdao, Shan dong, China
E-mail: dongyk@upc.edu.cn